RESPOSTAS EXERCÍCIO DE RR1

1-

O recurso de **interface** em Java serve como um "contrato" ou um "plano" (blueprint) para uma classe. Ela define um conjunto de métodos que uma classe *deve* implementar, mas não diz *como* ela deve implementá-los.

Essencialmente, a interface define **o que** uma classe deve ser capaz de fazer, sem se preocupar com **como** ela fará.

Cenários de uso:

- **Para permitir Polimorfismo:** Este é um dos usos mais importantes. Você pode programar sua aplicação para depender de uma interface, e não de uma classe concreta. Isso permite que você trate objetos de classes completamente diferentes da mesma maneira, desde que eles implementem a mesma interface.
  Exemplo (baseado na Questão 2): Você pode ter uma interface `FormaGeometrica` com um método `calcularArea()`. As classes `Circulo`, `Triangulo` e `Quadrado` implementariam essa interface, cada uma com sua própria fórmula. Seu código principal (o `main`) poderia então ter uma lista de `FormaGeometrica` e chamar `calcularArea()` em cada item, sem precisar saber se é um círculo ou um triângulo .
- **Para desacoplar o código:** As interfaces ajudam a reduzir a dependência entre diferentes partes de um sistema. O código que "consome" (usa) uma funcionalidade não precisa conhecer a implementação exata; ele só precisa conhecer a interface.
  Exemplo (baseado na Questão 5): Seu sistema pode depender de uma interface `RepositorioProduto`. Você pode ter uma implementação que usa arrays (`RepositorioProdutoArray`) e outra que usa `ArrayList` (`RepositorioProdutoArrayList`) . Se o resto do seu sistema usar apenas a interface `RepositorioProduto`, você pode trocar uma implementação pela outra sem quebrar o código.
- **Para simular "Herança Múltipla":** Em Java, uma classe só pode herdar (estender) de uma outra classe (herança simples). No entanto, uma classe pode *implementar* múltiplas interfaces. Isso permite que uma classe "assine" vários contratos e herde diferentes conjuntos de comportamentos abstratos.

2- A solução para a Questão 2 foi criar uma **interface** chamada `Forma`. Esta interface funciona como um contrato, definindo um único método obrigatório chamado `area()`.

Cada forma geométrica (`Circulo`, `Triangulo`, `Retangulo` e `Quadrado`) é uma classe separada que **implementa** a interface `Forma`. Ao fazer isso, cada classe é forçada a fornecer sua própria implementação (sobrescrever) o método `area()`, usando a fórmula matemática correta para si .

O ponto principal, visto no método `main` , é o uso de **polimorfismo**: criamos uma variável do tipo da interface (`Forma formaCalculavel`). Graças a isso, o `main` pode receber qualquer um dos objetos (seja um `Circulo` ou um `Triangulo`) e chamar o método `formaCalculavel.area()` sem precisar saber qual é o tipo específico da forma ou como ela calcula sua área. Isso atende perfeitamente ao requisito de manter o `main` independente dos detalhes de implementação .

3- As classes foram implementadas.

4-1. Análise: Elas possuem algo em comum? Existe replicação de código?

Sim, elas possuem quase tudo em comum.

As suas classes `RepositorioProdutoPerecivelArray` e `RepositorioProdutoNaoPerecivelArray` são um exemplo perfeito de **replicação de código** .

- Ambas possuem exatamente os **mesmos métodos**: `procurarIndice`, `existe`, `inserir`, `atualizar`, `remover` e `procurar`.
- Ambas possuem a **mesma lógica interna** para cada um desses métodos (incluindo os erros lógicos que discutimos).
- Ambas usam a **mesma variável de controle** `private int index = -1;`.

A **única diferença** real entre as duas classes é o tipo de array que elas gerenciam: uma usa `ProdutoPerecivel[]` e a outra usa `ProdutoNaoPerecivel[]`.

4-2. Solução: Seria possível ter apenas uma coleção? Em caso positivo, implemente.

Sim, é perfeitamente possível.

A solução é usar **Polimorfismo**, que é um dos principais pilares da Orientação a Objetos (e o mesmo conceito que você usou na Questão 2 com a interface `Forma`).

Como funciona:

1. Assumindo que `ProdutoPerecivel` e `ProdutoNaoPerecivel` são "filhos" (subclasses) de uma classe-mãe (superclasse) chamada `Produto`.
2. Criamos **uma única classe** de repositório (por exemplo, `RepositorioProdutoArray`).

3. Dentro dessa classe, o array interno não será de um tipo específico, mas sim do tipo-pai: `private Produto[] produtos;`.
4. Um array do tipo `Produto[]` pode armazenar *qualquer* objeto que "seja-um" `Produto`, o que inclui `ProdutoPerecivel` e `ProdutoNaoPerecivel`.
5. Os métodos (`inserir`, `atualizar`, `procurar`) passam a receber e retornar o tipo `Produto`.

5- A questão pergunta se é possível flexibilizar meu sistema para que eu possa usar diferentes tipos de repositórios, como trocar um que usa `Array` por um que usa `ArrayList` , e como eu faria isso .

Sim, é perfeitamente possível. A solução para isso é o conceito central do exercício: **usar Interfaces** .

O Problema: Acoplamento

Como meu código estava, eu tinha classes "concretas" como `RepositorioProdutoArray` (da Questão 4) e `RepositorioProdutoArrayList` (da Questão 6). Se minha classe principal (ou qualquer outra parte do sistema) precisasse de um repositório, eu teria que "amarrar" meu código a uma implementação específica.

Por exemplo: `RepositorioProdutoArray meuEstoque = new RepositorioProdutoArray(100);`

O problema aqui é que minha variável `meuEstoque` é do tipo `RepositorioProdutoArray`. Se amanhã eu decidir que o `RepositorioProdutoArrayList` é melhor (porque ele cresce sozinho e não tem os bugs de "buracos"), eu teria que "caçar" e mudar o tipo dessa variável em todos os lugares do meu código. Meu sistema está **fortemente acoplado** à implementação de Array.

A Solução: Programar para a Interface (Desacoplamento)

Para "flexibilizar" o sistema, eu preciso de um "contrato" que defina **o que** um repositório faz, sem se importar com **como** ele faz. Isso é exatamente o que uma interface faz .

Então, minha implementação seria:

1. **Eu criaria uma interface** chamada, por exemplo, `RepositorioProduto`.
2. Dentro dessa interface, eu definiria as assinaturas de todos os métodos públicos que um repositório deve ter: `inserir(Produto p)`, `remover(int codigo)`, `procurar(int codigo)`, `atualizar(Produto p)` e `existe(int codigo)`.
3. Em seguida, eu faria minhas duas classes concretas **implementarem** essa interface. (O Eclipse tem uma ferramenta de refatoração chamada "Extract Interface" que pode fazer isso automaticamente).

    - `public class RepositorioProdutoArray implements RepositorioProduto { ... }`
    - `public class RepositorioProdutoArrayList implements RepositorioProduto { ... }`

O Resultado: Flexibilidade Total

Depois de fazer isso, minha classe principal não precisa mais saber qual é a implementação concreta. Eu posso usar **Polimorfismo**:

Eu declaro minha variável usando o tipo da **interface**: `RepositorioProduto meuEstoque;`

E na hora de criar o objeto, eu posso **escolher** qual implementação eu quero, sem que o resto do meu código perceba a diferença:

- **Quero usar Array?** `meuEstoque = new RepositorioProdutoArray(100);`
- **Mudei de ideia, quero usar ArrayList?** `meuEstoque = new RepositorioProdutoArrayList(100);`

O resto do meu código (que chama `meuEstoque.inserir(p)` ou `meuEstoque.procurar(123)`) continua funcionando perfeitamente, porque ele só "sabe" que `meuEstoque` é um `RepositorioProduto` e que cumpre o contrato da interface.

É assim que eu torno o sistema flexível e permito que diferentes implementações de repositório sejam usadas .

6- A classe RepositorioProdutoArrayList foi implentada.